GKSS 2000/37

Feedforward–Backpropagation Neural Net Program `ffbp1.0`

H. Schiller

Feedforward–Backpropagation Neural Net Program `ffbp1.0`

Helmut Schiller

Abstract

`ffbp1.0` is a C-program for training of feedforward backpropagation neural nets (NN) running on UNIX computers. The program can be run in batch mode but also allows user intervention at any time.

A given table can be used as training&test–sample to solve different problems, like generation of a NN emulating a forward model as well as generation of a NN emulating its inverse.

NN's trained with `ffbp1.0` are stored in a file. Interfaces to use such NN files exist for `C, MATLAB` and `pvwave/IDL`.

As a byproduct of restricting to multilayer feedforward–backpropagation NN's the program is about twice as fast as the Stuttgart Neural Network Simulator `SNNS 3.1`.

A `zip`–file containing all sources, java classes and `html`–documentation can be obtained from `http://gfesun1.gkss.de/software/ffbp`. A `README`–file explains the installation.

Feedforward–Backpropagation Neuronales Netz Programm `ffbp1.0`

Abstract

`ffbp1.0` ist ein auf UNIX–Rechnern lauffähiges C-Programm zum Training von (Feedforward Backpropagation) Neuronalen Netzen (NN). Das Programm arbeitet im Batchmodus, aber der Nutzer kann jederzeit eingreifen.

Eine als Trainings&Test–Sample dienende Tabelle kann zur Lösung verschiedener Probleme genutzt werden, beispielsweise zur Erzeugung eines NN welches ein Vorwärtsmodell emuliert oder aber zur Erzeugung eines das inverse Modell emulierenden NN's.

Mit`ffbp1.0` trainierte NN's werden in einem File abgelegt. Die Möglichkeit der Nutzung solcher NN files besteht für `C, MATLAB` und `pvwave/IDL`.

Durch die Einschränkung auf Multilayer Feedforward–Backpropagation NN's ist das Program etwa zweimal schneller als der 'Stuttgart Neural Network Simulator' `SNNS 3.1`.

Ein `zip`–File mit allem Quellcode, Java Klassen und `html`–Documentation kann bei `http://gfesun1.gkss.de/software/ffbp` bezogen werden. Ein `README`–File erklärt die Installation.

Contents

1 User documentation **7**

1.1 Usage of the program . . . 7
1.2 Preparation of files . . . 8
1.2.1 Pattern related files . . . 8
1.2.2 The `ffbp1.0.start` file . . . 9
1.2.3 The `ffbp1.0.steer` file . . . 10
1.3 Running the program `ffbp1.0` . . . 11
1.3.1 The log files . . . 11
1.3.2 Communication with the running program . . . 11
1.3.3 The `.net` files . . . 12
1.4 The `ff4plot1.0` utility . . . 14
1.5 Usage of trained NN (without the `ffbp1.0`-package) . . . 14
1.5.1 The `C` interface . . . 14
1.5.2 The `MATLAB` interface . . . 15
1.5.3 The `pvwave/IDL` interface . . . 16
1.6 Summary of program requirements . . . 18

2 Technical documentation **19**

2.1 Data structures . . . 19
`anchor` . . . 19
`patterns` . . . 19
`parameters` . . . 20
`feedforward` . . . 20
`backprop` . . . 21
`backprop_part` . . . 21
2.2 Procedures . . . 21
`ffbp1.0` . . . 21
`**make_vecv(long n, long *s)` . . . 21
`***make_mtxv(long n, long *s)` . . . 21
`alloc_ff(parameters p)` . . . 21
`make_ff_from_file(char *filename)` . . . 22
`alloc_bp(feedforward ff)` . . . 22
`alloc_bp_part(feedforward ff, feedforward old)` . . . 22
`make_anchor()` . . . 22
`make_patterns(anchor a)` . . . 22
`ff_ini(feedforward ff, parameters p)` . . . 23
`*make_defaults(anchor a, patterns p)` . . . 23
`ffbp_newseq(feedforward ff, backprop bp)` . . . 23
`ffbp_newseq_part(feedforward ff, backprop_part bpp)` . . . 23
`ff_proc(feedforward ff)` . . . 23
`ff_to_file(feedforward ff, FILE *fp)` . . . 23

ff_save(anchor a, feedforward ff, patterns p) 23
ff_error(feedforward ff, double **in, double **out, long np) 24
*ff_gen_name(feedforward ff, double error) 24
ff_put_into(feedforward from, feedforward to) 24
ffbp_report(feedforward ff, patterns p, long nc, boolean b) . 25
bp_proc(feedforward ff, backprop bp, parameters p) 25
bp_proc_part(feedforward ff,backprop_part bpp,parameters p) . 25
2.3 Utilities 25
myallocs.c 25
make_alphatab() 25
prep_shufind(anchor a, patterns p) 25
make_shufind(anchor a, patterns p) 26
get_pars(parameters *p) 26
get_user_cmd() 26
new_steer_file(anchor a, feedforward ff, patterns p) 26
scp(double *x, double *y, long n) 26

1 User documentation

`ffbp1.0` is a C-program for training of feedforward backpropagation neural nets (NN) running on UNIX computers. The learning function implemented is the gradient descent algorithm with momentum term as descibed in many text books, *e.g.* [Bishop 1996]. The program meets the following design goals:

- user interaction with the program can be started from any terminal which has access to the computer (start the program in the institute and interact from home). If the terminal allows for XWindows a java program supplies a graphical user interface (GUI) for user intervention.
- the program can be run with no user interaction at all
- only one pattern file is needed for training and testing. Which points (= lines of the pattern file) are used for training and testing, respectively, is fixed in a `.dscp` file.
- the pattern file is organized by named columns (each column giving one variable). Which variables are used for input to the net and output from the net, respectively, is fixed in a `.usage` file. Therefor the same pattern file can be used to *e.g.* create a net for the forward and a net for the inverse model or to create different versions of an inverse model specifying different subsets of the variables in the respective `.usage` file.
- allow for starting with a small net which can be enlarged later if necessary, *i.e.* the number of neurons in hidden layers can be increased as well as the number of hidden layers.

As a byproduct of restricting the program to multilayer feedforward backpropagation neural nets the program is about twice as fast compared with the Stuttgart Neural Network Simulator `SNNS 3.1` [SNNS 1995].

1.1 Usage of the program

For each problem an extra directory has to be used (let it be `myproblem1`). The term 'problem' here summarizes all the nets derived from a fixed data set and using the same columns of the data set for net input/output.In `myproblem1` two subdirectories have to be created: `nets` (where the NN's are stored) and `msg` (used for user interference). In `myproblem1` one should have the following programs:

`expand_dscp` to generate from a basic description of the data set a complete one as needed for the given data set.

`ffbp1.0` to create and train the NN's for the problem.

`ff4plot1.0` which can be used to calculate NN output for a given net from the test sample

`java` classes which establish a graphical user interface to `ffbp1.0`.

1.2 Preparation of files

In this subsection the necessary input files are described. All input files are expected in the directory `myproblem1`.

1.2.1 Pattern related files

The pattern file is expected to have the `.patt` extension. Each point (=pattern) has the same number of variables and is stored as a line in the `.patt` file. The name of this file is used as the name of the 'problem' (let it be `probl1`). The user then has to generate two files: `probl1.dscp` and `probl1.usage`.

`probl1.dscp`
gives the names of the variables in the pattern file followed by a line containing 'end':

`var-name`$_1$

`var-name`$_2$

...

...

`var-name`$_k$

`end`

The description file is completed by running the program `expand_dscp`:

```
expand_dscp probl1 ntrain
```

where `ntrain` is the number of patterns which will be used for training. As the selection of points for training (the remaining patterns go to the test sample) is done randomly this number `ntrain` is only met approximately.

`probl1.usage`
gives the information about the variables used for NN input and output, respectively. The names of the variables refer to those given in the `.dscp` file:

`in` (number of input neurons)

`var-name_i`$_1$

var-name_i$_2$

...

var-name_i$_{in}$

out (number of output neurons)

var-name_o$_1$

var-name_o$_2$

...

var-name_o$_{out}$

1.2.2 The ffbp1.0.start file

The file ffbp1.0.start contains general informations which concern the whole problem and the NN architecture at the program start:

ffbp1.0.start contains

name_of_the_problem

ns nc

errlim

first_net

The values ns, nc concern the so called shuffling. Shuffling means that the order in which the patterns are presented to the net during the training changes from iteration to iteration. Since the generation of a random index list is rather time consuming in ffbp1.0 a fixed number ns of such random index arrays is generated and used for nc iterations (then new arrays are generated). The actual index array for each iteration is choosen randomly from the ns arrays.

The errlim is a lower limit of the error function at which the training will stop.

The specification of a net like first_net needs only information about the hidden planes (as NN input/output is fixed for a given problem by definition). The convention is to indicate the NN size in the following format:

h1xh2x...xhk

where we have k hidden planes with hi neurons in the i–th hidden plane (for a NN with only one hidden plane no x is given).

ffbp1.0.start Example:

```
probl1
100 50000
1.e-8
30x10
```

1.2.3 The `ffbp1.0.steer` file

Before `ffbp1.0` is started a last file has to be prepared: `ffbp1.0.steer`. It contains commands steering the run (the file must exist, but it could be empty: then the default parameters are used). The commands are supplied in a line oriented fashion: each line starts with a letter indicating the command followed by up to two parameters. Commands are read until iterations (`i` or `j`) are asked for. Then the parameters read so far are used for these iterations. If the iterations are done the next portion of the `ffbp1.0.steer` file is read. The order of commands within one such portion does not matter: the order in which they will be applied by the program is fixed (see below). If the file is exhausted the program goes into the wait state (`p` command is executed).

The valid commands are:

letter command	parameter(s)	comment
`b`	`b_low b_high`	Biases are generated in `[b_low,b_high]`
`e`	`size`	enlarge the net to `size`
`i`	`# iterations`	Learn all net parameters
`j`	`# iterations`	Learn only new net parameters
`l`	`learn_rate`	
`m`	`momentum`	
`p`		Pause, wait for user
`q`		Quit
`r`	`name.net`	Restart from net in `name.net`
`s`		Save actual net
`t`	`threshold`	Save actual net if `error lt threshold*error_last_saved`
`v`	`k`	print to `ffbp1.0.loge` file every `k` iterations
`w`	`w_low w_high`	Weights are generated in `[w_low,w_high]`
`x`	`neglerr`	Neglect errors less then `neglerr`

The commands (if given) are executed in the order `s q r e p i|j` with parameters (if applicable) as supplied by the commands or from defaults. The default values are

command	parameter(s)
b	-1 1
i	1000
l	0.6
m	0.2
t	0.8
v	100
w	-1 1
x	0

So with one portion of the steer file one can *e.g.* save the last net, enlarge this one or an old one, initialize its new net parameters and then iteratively change all or only the new net parameters.

1.3 Running the program `ffbp1.0`

If started first time the program changes the permissions for the `.patt, .usage, .dscp, .start`–files to 'read only', to assure the consistency of these files. Also two log files are created, to which output of later runs is appended.

1.3.1 The log files

There are two log–files which are flushed after each printing so they exhibit always the most recent state.

`ffbp1.0.loge` contains pairs of iteration-number and error. The iteration number refers to the respective program run. The error is derived from a pass through the training sample with all learning parameters set to zero. The error given is the sum over the training sample of the sum over the output neurons of the squared error divided by the number of output neurons and by the number of points in the training sample.

`ffbp1.0.logmsg` records all parameter changes, names of saved net files etc.

1.3.2 Communication with the running program

The file `working` exists during the program run in the directory `msg`. There also the file `old` is created at the program start. These two files are the basis for the user↔program

communication. The communication can either be performed 'by hand' or through a graphical user interface.

Communication 'by hand'

If the user renames the `old` file

```
mv msg/old msg/halt
```

the program will pause after the current iteration is finished and the user can examine the log files (the last line of `ffbp1.0.logmsg` should read '`user interferes`'). Now the user can decide either to continue the run

```
mv msg/halt msg/cont
```

or to create a new steer file `ffbp1.0.steer` and supply it by

```
mv msg/halt msg/new
```

The program renames this file again to `old` and the next communication can start.

If the program runs into a `p` command or exhausts the `ffbp1.0.steer` file the last line in `ffbp1.0.logmsg` will read '`waiting for user commands`'. Then the program can be halted

```
mv msg/old msg/halt
```

and be given a new `ffbp1.0.steer` file.

Communication by graphical user interface (GUI)

To ease the communication with the program and the inspection of the log files as well as the creation of new steer files a GUI relying on `java` was built. Like the communication 'by hand' it can start at any time

```
java Ffbp1v0_ap
```

which, if `java` is included in the shell variable `PATH`, will lead to a window like shown in fig. 1.

If the GUI was not started from the problem-directory using `File->Open` one can change to the problem directory by selecting any file therein. `Help->Explain` gives a window which helps to prepare the steer file. The `examine` button displays the log files. The `halt ffbp` button becomes disabled when pressed and the possible further choices become enabled.

The GUI informs the user by dialog windows which can be closed by a mouse click into them.

1.3.3 The `.net` files

The actual net is saved

- if a `s` command is met,
- if the program is halted ,

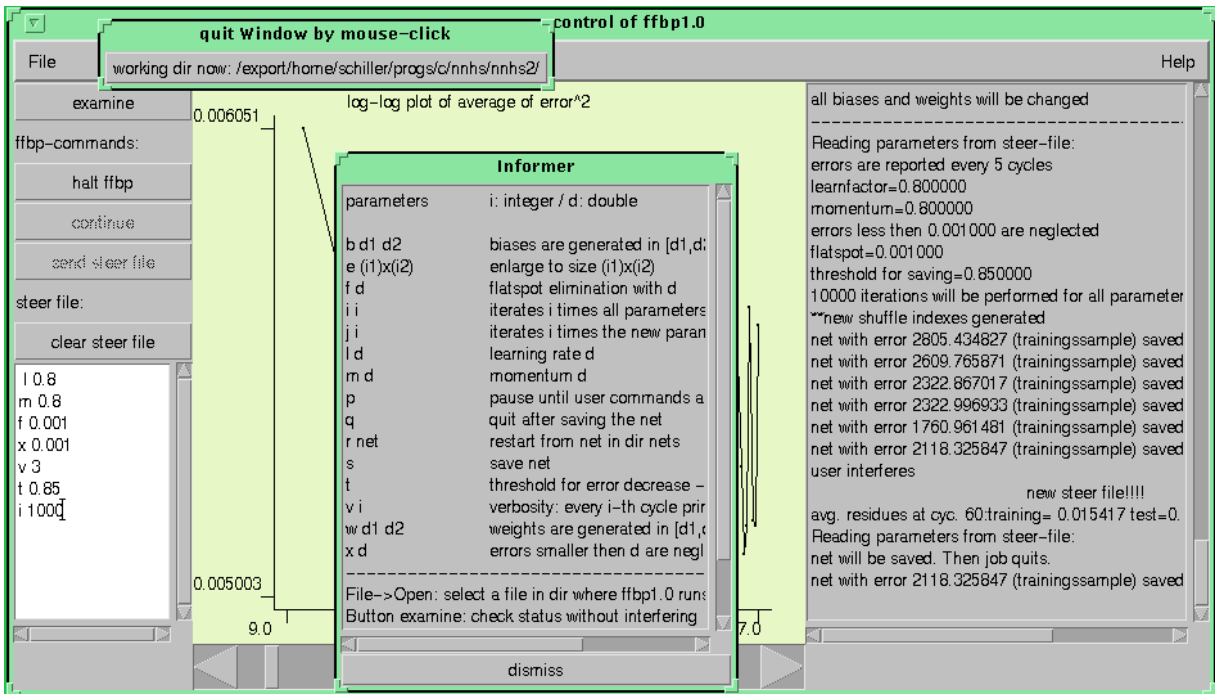

quit Window by mouse-click
control of ffbp1.0
File
working dir now: /export/home/schiller/progs/c/nnhs/nnhs2/
Help
examine
ffbp-commands:
halt ffbp
continue
send steer file
steer file:
clear steer file
l 0.8
m 0.8
f 0.001
x 0.001
v 3
t 0.85
i 1000
log-log plot of average of error^2
0.006051
0.005003
9.0
7.0
Informer
parameters i: integer / d: double
b d1 d2 biases are generated in [d1,d
e (i1)x(i2) enlarge to size (i1)x(i2)
f d flatspot elimination with d
i i iterates i times all parameters
j i iterates i times the new paran
l d learning rate d
m d momentum d
p pause until user commands a
q quit after saving the net
r net restart from net in dir nets
s save net
t threshold for error decrease –
v i verbosity: every i–th cycle prir
w d1 d2 weights are generated in [d1,
x d errors smaller then d are negl
File–>Open: select a file in dir where ffbp1.0 runs
Button examine: check status without interfering
dismiss
all biases and weights will be changed
Reading parameters from steer–file:
errors are reported every 5 cycles
learnfactor=0.800000
momentum=0.800000
errors less then 0.001000 are neglected
flatspot=0.001000
threshold for saving=0.850000
10000 iterations will be performed for all parameter
**new shuffle indexes generated
net with error 2805.434827 (trainingssample) saved
net with error 2609.765871 (trainingssample) saved
net with error 2322.867017 (trainingssample) saved
net with error 2322.996933 (trainingssample) saved
net with error 1760.961481 (trainingssample) saved
net with error 2118.325847 (trainingssample) saved
user interferes
new steer file!!!!
avg. residues at cyc. 60:training= 0.015417 test=0.
Reading parameters from steer–file:
net will be saved. Then job quits.
net with error 2118.325847 (trainingssample) saved

Figure 1: The GUI for `ffbp1.0`

- before quitting.

The name of the netfile is combined from the netsize and the error (here error is the total error neither divided by the size of the training sample nor by the number of output neurons). The extension is `.net`.

The `.net` files have three sections:

1. general information for the user (see page 23):
   - the complete path and name of the problem,
   - the date of generation,
   - errors from the training and test sample, respectively and
   - the names and ranges of the variables which are input/output.
2. starting with '#' informations about the ranges from which the variables are [0,1]-transformed when supplied to the net
3. starting with '$' informations about the structure of the net and the values of the biases and weights. This is the only section used when restarting a problem with `ffbp1.0`. The structure of this section can be understood from the code shown at page 24.

The last two sections are read by programs which use the trained NN for their respective purpose.

1.4 The `ff4plot1.0` utility

When the final goal of the exercise – a low error function of the trained net – is met one will be interested to compare in the test sample NN output with the prescribed values. For this one can with

```
ff4plot1.0 netfile plotdata
```

generate a `plotdata` file with one line for each point of the test sample. The first values in a line are the NN input. Then for each output neuron a pair (NN output, true (= prescribed) value) follows. The `plotdata` file can be used to visualize the NN performance (already during NN training) like in fig. 2 using any available plot program.

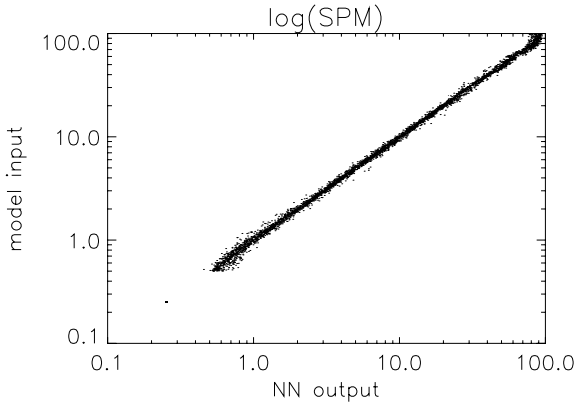


Figure 2: Visualization of `ff4plot1.0` output

1.5 Usage of trained NN (without the `ffbp1.0`-package)

At the time being there are `C, MATLAB` and `pvwave/IDL` interfaces to use NN's generated with `ffbp1.0`. To write a new interface one should consult the description of `ff_save` at page 23.

1.5.1 The `C` interface

The use of a trained NN is accomplished by the two `C`–functions `ff_work_prepare` and `ff_work` which should be used as follows:

```
long    *size;

..initialization

size=ff_work_prepare(net_file_name);
/* size[0]=# of NN inputs, size[1]=# of NN outputs */

while(more_points_to_process) {
        ..
        /*  we stored the NN input in the double array nn_in
            and are ready to feed it forward through the NN  */

        ff_work(nn_in, nn_out);

        /* now the NN output is stored in the
             double array nn_out  */
        ..

}
```

A complete example `ff_work_scheme.c` of a program using these two functions can be found in the `zip` distribution file. The following commands could be used to compile and run this program:

```
cc -xO5 ff_work_scheme.c ff_work.o -o ff_work_scheme -lm
ff_work_scheme  50x12_652.1.net mcrunshort.patt 50x12_652.1.res
textedit  50x12_652.1.res
```

1.5.2 The MATLAB interface

For `MATLAB` the class `nnhs` interfaces to NN's generated with `ffbp1.0`, therefor the following M–files must be put into the (new) directory `@nnhs`:

`nnhs.m:` `nn = nnhs(ffbp_file)` is the 'feedforward neural net' class constructor (from `ffbp1.0 .net file`),

`char.m:` `s = char(nn)` converts `nn` to `char`,

`display.m:` `display(nn)` does the command window display of neural net `nn`,

`in_nnhs.m:` `[in, inrange] = in_nnhs(nn)` returns the number of input neurons and their ranges,

`out_nnhs.m:` `[out, outrange] = out_nnhs(nn)` returns the number of output neurons and their ranges,

`ff_nnhs.m: out = ff_nnhs(nn, in)` does the evaluation of neural net response.

The demo program

```
mynn=nnhs('10_0.2.net');
% the NN '10_0.2.net' has two input and one output neuron
[x, y]=meshgrid(-4:.16:4);
z=zeros(51,51);
for l=1:51
        for m=1:51
                z(l,m)=ff_nnhs(mynn, [x(l,m) y(l,m)]);
        end
end
meshc(x, y, z);
```

produces the output shown in fig. 3.

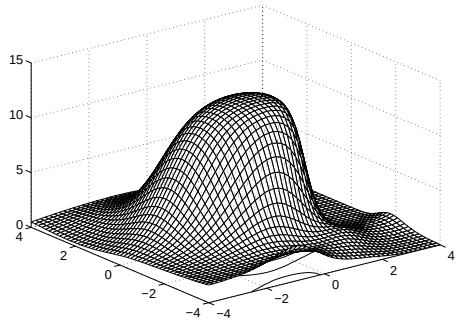


Figure 3: `ffbp1.0` NN used in above `MATLAB` program

1.5.3 The `pvwave/IDL` interface

Here the NN calculations are done by the process `calc2_NN`, which communicates via `stdio` with the mother process. `calc2_NN` is initialized with the name of the NN and responds with the number of input and output neurons of the given NN. Then in an endless loop it accepts inputs to the NN and responds with the resulting output values. The following example can be used to develop programs using NN's:

```
pro calc2_NN
common calc2_NN, NNlun, NN_input, NN_output
writeu, NNlun, NN_input
readu, NNlun, NN_output
end

pro demo_calc2_NN
common calc2_NN, NNlun, NN_input, NN_output; communicate with calc2_NN
```

```
NN_name='10_0.2.net'; which net to use

SPAWN, ['calc2_NN', NN_name], Unit = NNlun, /Noshell; start the program
NN_in =0L; the C-program needs long's
NN_out=0L

readu, NNlun, NN_in
readu, NNlun, NN_out
print, NN_name,' has ',NN_in,' input and ',NN_out,' output neurons'
NN_input=dblarr(NN_in);   must be double
NN_output=dblarr(NN_out); must be double

nx=51;  now we can use the NN
ny=51
x=8.*dindgen(nx)/(nx-1)-4.
y=8.*dindgen(ny)/(ny-1)-4.
z=dblarr(nx,ny)
for i=0, nx-1 do begin
    for j=0, ny-1 do begin
        NN_input(0)=x(i)
        NN_input(1)=y(j)
        calc2_NN;        here we use the NN
        z(i,j)=NN_output(0)
    endfor
endfor

free_lun, NNlun; we dont need the NN anymore

surface,z,x,y,xtitle='r!d1!n',ytitle='r!d2!n',ztitle='c',$
title='err1 '+NN_name,charsize=2.5

return
end
```

which gives the result shown in fig. 4.

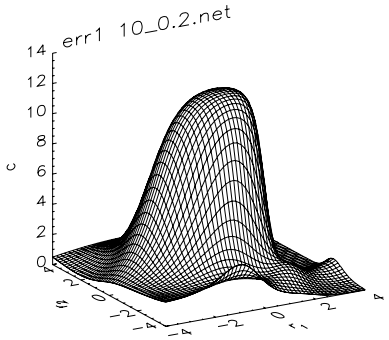


Figure 4: `ffbp1.0` NN used in above `pvwave` program

1.6 Summary of program requirements

The following overview summarizes the program requirements:

`dir myproblem1`

input files to be prepared:

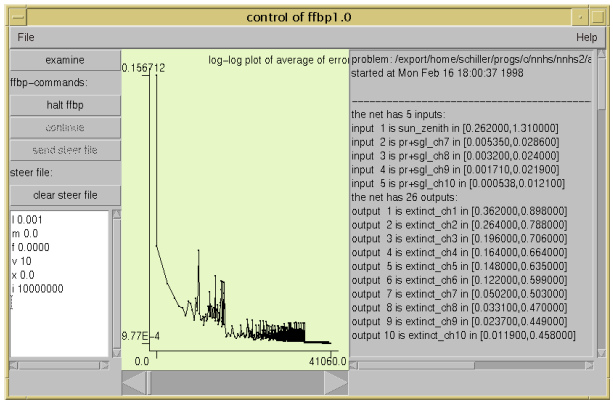
- The pattern file, *e.g.* `probl1.patt`, generated by a model run or from empirical data; to be used as training/test sample for the NN generation.
- The description file, *e.g.* `probl1.dscp`, with the names of the variables in the pattern file. This file must be completed using `expand_dscp`.
- The usage file, *e.g.* `probl1.usage`, defining the NN I/O.
- The `ffbp1.0.start` file containing informations about the whole problem and the NN architecture at the program start.
- The `ffbp1.0.steer` file with `ffbp1.0` commands to steer the NN training.

output files to be inspected during NN training:

- The file `ffbp1.0.loge` contains pairs of iteration-number and error.
- The file `ffbp1.0.logmsg` records all all changes of command parameters, names of saved net files *etc.*.

`dir msg` contains the files necessary to steer the user ↔ program communication

- 'by hand' or
- using the java based GUI



⇑ `ffbp1.0.steer` ⇑ `ffbp1.0.loge` ⇑ `ffbp1.0.logmsg`

`dir nets` contains the NN's saved during NN training

- which can be used during NN training to visualize the NN performance.
- the final NN can be used by `C, MATLAB, pvwave/IDL` programs for respective purpose.

2 Technical documentation

This section has the following three subsections: the first explains the data structures, the second gives details about the NN specific functions and in the third subsection functions of general interest are explained.

2.1 Data structures

The data structures are declared in `ffbpnn.h`

anchor
essentially stores the stuff from the `ffbp1.0.start` file

```
typedef struct {
        char            *problem;       /* name of the problem  */
        char            *pwd;           /* working directory    */
        boolean          newproblem;
        long             nhidden;       /* #of hidden pl. of the net
                                           in .start*/
        long            *hsize;         /* and their sizes  */
        long             n_shuffle;     /* #of shuffle index arrays     */
        long             new_shuffle;   /* #of cycl until new shuffleind
                                           are made */
        double           errlim;        /* lower limit, program stops
                                           if reached */
}               anchor;
```

patterns
contains informations about the training and test samples

```
typedef struct {
        long             npatterns;     /* #of patterns in .patt-file   */
        long             nvars;         /* #of variables in .patt-file  */
        char           **varname;       /* the names of the variables   */
        double          *min;           /* their minimum values */
        double          *max;           /* their maximum values */

        /* tr_in, tr_out, te_in, te_out: [pattern][variable]    */

        long             ntrain;        /* #of patterns for training    */
        double         **tr_in;         /* training input to net        */
        double         **tr_out;        /* desired net output for tr_in */

        long             ntest;         /* #of patterns for test        */
        double         **te_in;         /* test input to net            */
        double         **te_out;        /* desired net output for te_in */

        long             nnin;          /* #of input neurons    */
        long             nnout;         /* #of output neurons   */
```

```
        long           *index_in;       /* indexes of net input values
                                           wrt .patt  file */
        long           *index_out;      /* indexes of net output values
                                           wrt .patt file */
}               patterns;
```

parameters

are set by the last read portion of the `ffbp1.0.steer` file. 'portion' means commands until an i or j command is issued

```
typedef struct {
        double          learn;          /* learning rate */
        double          momentum;       /* momentum of last change */
        double          flatspot;       /* added to derivativ   */
        double          neglect;        /* errors with fabs() less then
                                           neglect dont matter */
        double          biasl;          /* next new biases will be gene-*/
        double          biash;          /* rated in [biasl,biash]       */
        double          wgtl;           /* next new weights will be gene-*/
        double          wgth;           /* rated in [wgtl,wgth]         */
        double          threshold;      /* save if threshold*last_save_err
                                           gt error */
        long            silence;        /* error reporting every
                                           silence iteration*/
        long            iter;           /* #of iterations to be performed */
        long            nplanes;        /* #of planes in new net        */
        long           *plsize;         /* their sizes  */
        boolean         save;           /* save command issued  */
        boolean         newsize;        /* new net size is specified    */
        boolean         all;            /* true if all parameters
                                           to be iterated */
        boolean         restart;        /* start from stored net        */
        boolean         ready;          /* quit command issued  */
        boolean         wait;           /* p command or .steer
                                           file exhausted */
        char            netfile[256];   /* name of net file for restart */
}               parameters;     /*from last .steer file portion */
```

feedforward

is the actual net feeding forward an input pattern

```
typedef struct {
        long            nplanes;        /* #of planes in net    */
        long           *size;           /* their sizes  */
        double       ***wgt;            /* weight
                                           [plane][to_neuron][from_neuron] */
        double        **bias;           /* [plane-1][neuron]    */
        double        **act;            /* neuron output[plane][neuron] */
        double         *input;          /* input[neuron]=act[0][neuron] */
        double         *output;         /* output[neuron]=
                                                  act[nplanes-1][neuron] */
```

```
}               feedforward;
```

backprop

is the actual net (minus feedforward) backpropagating the errors of an net output and changing all biases and weights

```
typedef struct {
        double        **ldbias;         /* last changes of **bias       */
        double       ***ldwgt;          /* last changes of ***wgt       */
        double        **delta;          /* [pl][neuron], pl=0..nplanes-2,
                                           backprop. error */
        double         *error;          /* errors[neuron] in output layer */

}               backprop;
```

backprop_part

is the actual net (minus feedforward) backpropagating the errors of an net output and changing biases and weights which did not exist before the enlargement of the net

```
typedef struct {
        double        **ldbias;         /* last changes of **bias       */
        double       ***ldwgt;          /* last changes of ***wgt       */
        double        **delta;          /* [pl][neuron], pl=0..nplanes-2,
                                           backprop. error */
        double         *error;          /* errors[neuron] in output layer */
        long           *old_size;       /* sizes of planes
                                           before enlargement */
}               backprop_part;
```

2.2 Procedures

ffbp1.0

is the main program

double **make_vecv(long n, long *s)

generates a vector $[i = 0, \ldots n-1]$ of n vectors of sizes $s[i]$ as needed for the biases.

double ***make_mtxv(long n, long *s)

generates a vector $[i = 0, ..n-2]$ of matrices of sizes $[s[i+1], s[i]]$ as needed for the weights connecting the layers.

feedforward alloc_ff(parameters p)

generates `feedforward` as described in parameters to describe a feedforward NN.

`feedforward make_ff_from_file(char *filename)`

reads a feedforward NN description from the file `filename`.

`backprop alloc_bp(feedforward ff)`

generates what in addition to `ff` is necessary to implement the backpropagation of errors and the adaption of the parameters of the NN.

`back_prop_part alloc_bp_part(feedforward ff, feedforward old)`

generates what in addition to `ff` is necessary to implement the backpropagation of errors and the adaption of the parameters of the NN which were not existing in the `old` NN.

`anchor make_anchor()`

- creates the files `msg/working` and `msg/old` for user communication,
- reads the file `ffbp1.0.start`,
- opens the files

pointer	filename	content
`fpsteer`	`ffbp1.0.steer`	commands steering `ffbp1.0`
`fploge`	`ffbp1.0.loge`	iteration# error
`fplogmsg`	`ffbp1.0.logmsg`	protocol
`fppattv`	*problem*`.patt`	patterns
`fppdscp`	*problem*`.dscp`	description of patterns
`fppusage`	*problem*`.usage`	usage of patterns

,

- changes the permissions of the files `ffbp1.0.start`, *problem*`.patt`, *problem*`.dscp`, *problem*`.usage` to read only.

`patterns make_patterns(anchor a)`

- allocates space for the parts of the patterns to be used for NN input/output (training+testing),
- stores the $(0, 1)$–scaled `ntrain` NN input vectors of length `nnin` to `tr_in[ntrain][nnin]`,
- stores the $(0, 1)$–scaled `ntrain` NN output vectors of length `nnout` to `tr_out[ntrain][nnout]`,
- stores the $(0, 1)$–scaled `ntest` NN input vectors of length `nnin` to `te_in[ntest][nnin]`,
- stores the $(0, 1)$–scaled `ntest` NN output vectors of length `nnout` to `te_out[ntest][nnout]`,
- closes `fppattv, fppusage, fppdscp`.

`void ff_ini(feedforward ff, parameters p)`

initializes the weights and biases of the `ff` NN in ranges given in `parameters`.

`parameters *make_defaults(anchor a, patterns p)`

- puts default values for parameters which can be changed with commands in the `ffbp1.0.steer` file,
- takes the net size given in `ffbp1.0.start` as default.

`void ffbp_newseq(feedforward ff, backprop bp)`

zeroes the fields with the last changes of biases and weights, which are used to store the values for the momentum term.

`void ffbp_newseq_part(feedforward ff, backprop_part bpp)`

zeroes the fields with the last changes of new biases and new weights, which are used to store the values for the momentum term.

`void ff_proc(feedforward ff)`

processes the given NN input through the NN to produce the corresponding NN output.

`void ff_to_file(feedforward ff, FILE *fp)`

writes the NN `ff` to file `*fp`.

`void ff_save(anchor a, feedforward ff, patterns p)`

saves the NN `ff` to a file. The name of the file is given by `ff_gen_name`. The file contains helpful informations like:

```
problem: /export/home/schiller/progs/c/nnhs/nnhs2/mcrunst
saved at Fri Aug  8 05:24:24 1997

trainings sample has total sum of error^2=696.980276
average of residues:
 training 696.980276/45802/11=0.005072  test 173.181012/11438/3=0.005047
 ratio avg.train/avg.test=1.005046

the net has 11 inputs:
input  1 is sun_theta in [0.000574,1.309000]
input  2 is view_theta in [0.000000,0.715600]
input  3 is view_phi in [0.000141,3.142000]
input  4 is refl1 in [0.000052,0.172700]
input  5 is refl2 in [0.000061,0.178200]
input  6 is refl3 in [0.000076,0.159300]
input  7 is refl4 in [0.000086,0.124200]
input  8 is refl5 in [0.000108,0.084930]
input  9 is refl6 in [0.000068,0.025000]
input 10 is refl7 in [0.000049,0.016750]
input 11 is refl8 in [0.000023,0.008868]
```

```
the net has 3 outputs:
output  1 is log_conc_phy in [-5.807000,3.909000]
output  2 is log_conc_gelb in [-6.214000,0.693100]
output  3 is log_conc_min in [-3.506000,3.912000]
```

For later usage of the stored NN after '#' the number of input neurons and their ranges of input values are given followed by the number of output neurons and their ranges of output values, respectively. After a '$' the parameters of the actual NN follow. The structure of this part is seen from the following piece of code which writes the parameters:

```
fprintf(fp, "$\n#planes=%ld", ff.nplanes);
for (pl = 0; pl < ff.nplanes; pl++)
        fprintf(fp, " %ld", ff.size[pl]);
fprintf(fp, "\n");
for (pl = 0; pl < ff.nplanes - 1; pl++) {
        fprintf(fp, "bias %ld %ld\n", pl + 1, ff.size[pl + 1]);
        for (i = 0; i < ff.size[pl + 1]; i++)
                fprintf(fp, "%lf\n", ff.bias[pl][i]);
}
for (pl = 0; pl < ff.nplanes - 1; pl++) {
        fprintf(fp, "wgt %ld %ld %ld\n",
                pl, ff.size[pl], ff.size[pl + 1]);
        for (i = 0; i < ff.size[pl + 1]; i++) {
                for (j = 0; j < ff.size[pl]; j++)
                        fprintf(fp, "%lf\n",ff.wgt[pl][i][j]);
        }
}
```

`double ff_error(feedforward ff, double **in, double **out, long npatt)`

calculates the total sum of squared errors produced by `ff` if given the `npatt` NN inputs `in[npatt][nnin]` and prescribed outputs `out[npatt][nnout]`.

`char *ff_gen_name(feedforward ff, double error)`

generates a name for the file to which the actual NN `ff` is to be saved. The name is built by concatenating the NN size with the error:

`30x10_244.5.net`

is the name of the file into which the NN of size `30x10` producing an error 244.5 (using the training sample) will be saved.

`void ff_put_into(feedforward from, feedforward to)`

copies the biases and weights from the (smaller) NN `from` to the (larger) NN `to`. (The remaining new biases and weights have are initialized.)

`void ffbp_report(feedforward ff, patterns p, long nc, boolean both)`
reports to the file `ff1.0.logmsg` at iteration `nc` about the net performance. `both` has to be `TRUE` if not only the performance of the NN with the training sample but also with the test sample is to be reported.

`void ff_info(feedforward ff, FILE *fp)`
writes the plane sizes of NN `ff` to `*fp`.

`void bp_proc(feedforward ff, backprop bp, parameters p)`
does for a given pattern which was fed forward by `ff_proc` the backpropagation of the error and the corresponding change of all the biases and all the weights and stores those changes so they can be used in the momentum term.

`void bp_proc_part(feedforward ff, backprop_part bpp, parameters p)`
does for a given pattern which was processed forward by `ff_proc` the backpropagation of the error and the corresponding change of the new biases and the new weights and stores those changes so they can be used in the momentum term.

2.3 Utilities

`myallocs.c`
contains a set of space allocating functions which chose `myexit()` (removing `msg/working`) in case of failure:

$\mathcal{X}$`alloc(long n)` $\mathcal{X}=$	returns space for `n`
`mc`	`char`
`mcp`	`char *`
`l`	`long`
`lp`	`long *`
`d`	`double`
`dp`	`double *`
`dpp`	`double **`

`void make_alphatab()`
generates `N_ALPHA` entries in `alpha_tab[N_ALPHA]` tabulating the logistic function in the range `(ALPHA_ANF,-ALPHA_ANF)` being used by `alpha(x)` which does the actual interpolation.

`void prep_shufind(anchor a, patterns p)`
allocates space for `n_shuffle` index vectors to shuffle the `ntrain` patterns.

`void make_shufind(anchor a, patterns p)`

fills the `shuffle[n_shuffle][ntrain]` index vectors with a random permutation of $(0, 1, \ldots, ntrain - 1)$ to shuffle through the `ntrain` training patterns.

`void get_pars(parameters *p)`

reads commands from `ffbp1.0.steer` and sets parameters accordingly.

`void par_info(parameters p, long cyc, FILE *fp)`

writes at iteration `cyc` the actual parameters `p` to `*fp`.

`enum user_cmd get_user_cmd()`

waits for user intervention and returns `new` or `cont` if the user renames `msg/halt` to `msg/new` or `msg/cont`, respectively.

`boolean new_steer_file(anchor a, feedforward ff, patterns p)`

checks if the user renamed `msg/old` to `msg/halt` and returns `FALSE` immediately if the user did not rename. Otherwise it waits until user intervention is ready and returns `TRUE` if the user supplied a new steering file `ffbp1.0.steer`.

`double scp(double *x, double *y, long n)`

calculates the scalar product of `x[n]*y[n]`.

References

[SNNS 1995] SNNS, Stuttgart Neural Network Simulator, *User Manual, Version 3.1*, University of Stuttgart, Institute for parallel and distributed high performance systems (anonymous `ftp ftp.informatik.uni-stuttgart.de (129.69.211.2)`).

[Bishop 1996] Bishop, Chris; Neural networks for pattern recognition; Oxford University Press, New York; 1996.